LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-203037

A White Paper Prepared for the OpenMP Architectural Review Board on DMPL: An OpenMP Debugging Interface

James Cownie, John DelSignore, Jr., Bronis R. de Supinski and Karen Warren

March 8, 2004

A White Paper Prepared for the OpenMP Architectural Review Board on DMPL: An OpenMP DLL Debugging Interface

James Cownie[1], John DelSignore, Jr.[1], Bronis R. de Supinski[2], and Karen Warren[2]

[1] Etnus, LLC, 24 Prime Parkway,
Natick, Massachusetts 01760
{jcownie, jdelsign}@etnus.com
http://www.etnus.com/Products/TotalView/index.html
[2] Lawrence Livermore National Laboratory, P.O. Box 808, L-560,
Livermore, California 94551-0808*
{bronis, kwarren}@llnl.gov
http://www.llnl.gov/icc/lc/DEG/TV.html

March 8, 2004

Abstract. OpenMP is a widely adopted standard for threading directives across compiler implementations. The standard is very successful since it provides application writers with a simple, portable programming model for introducing shared memory parallelism into their codes. However, the standards do not address key issues for supporting that programming model in development tools such as debuggers. In this paper, we present DMPL, an OpenMP debugger interface that can be implemented as a dynamically loaded library. DMPL is currently being considered by the OpenMP Tools Committee as a mechanism to bridge the development tool gap in the OpenMP standard.

1 Introduction

OpenMP is a widely used parallel programming model for shared-memory multiprocessor (SMP) architectures [1], [2]. The OpenMP organization initially focused on language design and run-time support. This focus has been successful - OpenMP now provides relative ease in writing codes that efficiently utilize SMP architectures.

While the initial focus has been successful, key areas for usability still need to be addressed. In particular, accurate information needed for debugging and analyzing performance of OpenMP applications can be difficult to obtain. Even when the in-

* This work was partially performed under the auspices of the U.S. Department of Energy by University of California LLNLaboratory under contract W-7405-Eng-48. UCRL-TR-203037.

formation is provided, it is very difficult to present the information consistently with the semantics of the OpenMP programming model.

The OpenMP Tools Committee is working on addressing some of these difficulties. It has considered the POMP interface [3] as a standard method to instrument OpenMP applications and to gather performance data. However, the POMP interface does not address problems that arise when one attempts to debug an OpenMP code. This document presents DMPL (pronounced “dimple”), an interface for OpenMP debugger support that can be implemented as a dynamically loaded library (DLL). We recommend that each compiler vendor implement this library and provide it along with the OpenMP run-time library. Debuggers would then dynamically load this library to obtain the information needed for users to debug OpenMP codes.

The rest of this paper first explores key background issues for debugging OpenMP application codes. We then discuss why a DLL-based OpenMP debugger interface is the right solution. Next, we present the current definition of DMPL. We conclude with a discussion of why it is inappropriate to add the required debugger support to a performance instrumentation API and open issues for OpenMP debugger support, which may eventually be addressed in a revision of this white paper. Finally, the complete Dmpl.h header file is provided in the appendix.

2 Background

2.1 Multiple Compilers, User Levels

Debugging features required for OpenMP codes are similar to those required for sequential code. Users want to work with the "source-level" code and to plant breakpoints within OpenMP regions. They want to step and otherwise control execution in those regions and to examine variables within them. However, the underlying threading provided by the OpenMP compiler and run-time complicates these goals. Should the debugger present outlined routines that the compiler creates so that OpenMP regions can be executed by multiple threads? Does the user expect the same view for shared, private, and thread private variables?

Some users understand how the OpenMP directives accomplish the desired parallelism; other users don’t care about such details. Ideally, the debugger presents a serial code to the latter users while supporting a more detailed view that sophisticated users may desire. More realistically and practically, the debugger allows the user to see thread private variables and manipulate the threads individually. Regardless, a portable debugger must understand the output from various compilers (e.g., HP/Compaq, IBM, Intel-Guide, SGI and Sun) and present it to the user in a way that avoids involving the user in the underlying transformations.

2.2 Compiler Transformations

The OpenMP language specifications allow a variety of implementations. Portable debuggers must directly solve some possible implementation differences. However, a standard interface can assist the debugger in hiding many implementation details.

Depending on the compiler, the user's code may first be preprocessed and the resulting code actually compiled. The debugger deals with the preprocessed code. Since the user wants to debug the original code, the preprocessor must insert line number directives. The debugger must interpret the directives. Although line number directives are already standardized, some OpenMP implementations that use preprocessing have not always included them.

The compiler uses threads to achieve the necessary parallelism. Various threading packages can be used: pthreads, sproc, and other proprietary threading packages. The debugger user must be able to asynchronously control these threads. This goal means that the underlying thread package must include support to synchronize thread execution, to single step threads in lockstep and for thread-specific breakpoints. Although the debugger must handle the underlying thread package, OpenMP implementers can help ensure that the package includes needed support for asynchronous thread control.

To achieve the desired parallelism, the compiler constructs outlined routines that the master thread calls and the worker threads execute. There may be multiple outlined routines for a single worksharing construct. Calls to the outlined routines become part of the stack traces. The stack frames from the routines become part of the calling stack frame. Ultimately, what looks like straight-line code to the user isn't. The debugger needs a standard method to recognize outlined routines and to associate them with worksharing constructs. Similarly, the debugger needs a standard mechanism to identify run-time library routines, for which the names vary for every compiler. Given these mechanisms, the debugger can then eliminate the calls from stack frames and support the appearance of straight-line code, if the user desires it.

Each compiler mangles the source code names differently when it makes up names for the outlined routines and their variables. The debugger must demangle a mangled name in order to present the user with a recognizable name. Further, the debugger may need to determine language meaning for the mangled name, such as whether a variable is shared or private. The debugger also must use the proper addressing mode for the variable, which can vary substantially between compilers, particularly for thread private data. The OpenMP implementation must provide the debugger with a standard mechanism for name demangling and this related information.

3 DMPL Objective

DMPL is a dynamically loaded library interface that provides the debugger with information needed to support OpenMP applications. The TotalView parallel debugger [4] already uses this paradigm for debugging pthreads, MPI message queues [5], and UPC [6]. Similarly to run-time libraries, OpenMP compiler vendors should provide a

DMPL library. The debugger, which already contains information about the run-time parallel environment, will load the DLL.

This paper defines the DMPL interface, which separates the OpenMP implementation from the debugger. This interface must provide a debugger with all the information that it needs to present the user with a view of his code both in terms of the original source and with additional thread details. The routines supplied by the vendor in the DLL can be linked dynamically in the debugger allowing callbacks to the debugger. The DLL is an OpenMP implementation-specific product and its implementation details are left to the OpenMP implementer.

4 DMPL Interface

The proposed DLL is a two-way interface between the debugger and the DLL itself. When the debugger needs to display the value or address of a thread private object, it will make a call to a DLL function to extract the absolute address of the object. The DLL itself will make calls to the debugger to access information about the target process or thread. The DLL must not use global data internally because the debugger may be debugging independent processes simultaneously. Instead, it must associate data between calls with the specific object.

4.1 DMPL Types

DMPL includes several defined types in order to pass target architectural information to the debugger and to simplify interface function definitions. Other types provide function return codes, language values and codes for demangling information.

```
typedef struct {
    int short_size;
    int int_size;
    int long_size;
    int long_long_size;
    int pointer_size;                   /* sizeof (void *) */
} DMPL_target_type_sizes_t;
```

```
typedef unsigned long long              DMPL_taddr_t;
```

```
typedef long long                       DMPL_tword_t;
```

```
typedef struct _DMPL_process_t          DMPL_process_t;
typedef struct _DMPL_thread_t           DMPL_thread_t;
typedef struct _DMPL_type_t             DMPL_type_t;
```

```
typedef struct _DMPL_process_info_t     DMPL_process_info_t;
typedef struct _DMPL_thread_info_t      DMPL_thread_info_t;
```

Fig. 1. Types for DMPL Target Architectural Information

Figure 1 shows that target architectural information includes the sizes of pointers and integer types and address values. Since `DMPL_taddr_t` is an unsigned long

long, it allows for addresses on any architecture. Specifically, a 32-bit debugger should be able to debug a 64-bit target process.

```
enum {                              Explanation
    DMPL_ok=0,                      Success
    DMPL_tls_unallocated            No space allocated
    DMPL_name_too_long,             Buffer too small
    DMPL_name_unchanged,            Name wasn't demangled
    DMPL_first_user_code = 100      Allow more pre-defines
};
```

Fig. 2. DMPL Result Codes

Figure 1 also includes opaque types for process, thread and type information. The debugger and the DLL will each determine what process/thread information it needs. The types `DMPL_process_info_t` and `DMPL_thread_info_t` are specifically provided for the DLL. The opaque types, which will be cast to concrete types in the debugger or the DLL, preserve types across the DMPL interface. We use these undefined structures instead of void pointers in order to provide more compile-time checking at the cost of explicit casts in the library and support code.

```
enum {
    DMPL_lang_unknown     =     0,
    DMPL_lang_c           =     'c',
    DMPL_lang_cplus       =     'C',
    DMPL_lang_f77         =     'f',
    DMPL_lang_f9x         =     'F'
};
```

Fig. 3. DMPL Language Codes

We have included three enumerated types in DMPL. In order to avoid potential issues with different compilers implementing enumerated types as different sized objects, we actually use `int` when they are used as a result or parameter type. Most DMPL functions return one of the result codes listed in Figure 2. Although both the DLL and the debugger will use values starting with `DMPL_first_user_code`, calling context will eliminate any confusion. Figure 3 shows the DMPL language values, which support providing the debugger with language-specific information from the OpenMP run-time library.

```
enum {
    DMPL_varinfo_needs_dereference                  = 0x8000,
    DMPL_varinfo_none                               = 0,
    DMPL_varinfo_private,
    DMPL_varinfo_shared,
    DMPL_varinfo_firstprivate,
    DMPL_varinfo_lastprivate,
    DMPL_varinfo_firstlastprivate,
    DMPL_varinfo_reduction,
    DMPL_varinfo_threadprivate,
    DMPL_varinfo_threadshared,
    DMPL_varinfo_copyin
```

```
};
```

Fig. 4. DMPL Demangling Codes

The DLL uses DMPL demangling codes that are listed in Figure 4 to communicate scoping information to the debugger. `DMPL_varinfo_needs_dereference` should be 'or-ed' into the appropriate scoping code in order to indicate that dereferencing is necessary to access the variable.

4.2 DMPL Functions

In order to access the information needed for OpenMP codes, the debugger loads the DMPL DLL. The debugger then calls the DMPL function `DMPL_initialize`, which performs steps necessary to initialize the DLL, including instantiation of a DMPL callback table that supports communication between the DLL and the debugger. The rest of this section describes DMPL functions and function types, beginning with the DMPL callback table.

4.2.1 DMPL Callback Table

In order to provide the information needed by the debugger, the DLL must make calls to the debugger. This architecture supports a clean interface between the DLL and the debugger, avoiding the use of global data. The DLL uses functions contained in the DMPL callback table, as defined in Figure 5. The function types used to define the callback table are shown in Figure 6. The primary callback functions communicate process, thread and type information between the DLL and the debugger.

```
typedef struct {
    DMPL_put_process_info_ft     DMPL_put_process_info_fp;
    DMPL_get_process_info_ft     DMPL_get_process_info_fp;
    DMPL_put_thread_info_ft      DMPL_put_thread_info_fp;
    DMPL_get_thread_info_ft      DMPL_get_thread_info_fp;
    DMPL_get_process_ft          DMPL_get_process_fp;
    DMPL_get_type_sizes_ft       DMPL_get_type_sizes_fp;
    DMPL_find_symbol_ft          DMPL_find_symbol_fp;
    DMPL_find_type_ft            DMPL_find_type_fp;
    DMPL_get_data_ft             DMPL_get_data_fp;
    DMPL_get_pthread_key_ft      DMPL_get_pthread_key_fp;
    DMPL_target_to_host_ft       DMPL_target_to_host_fp;
    DMPL_malloc_ft               DMPL_malloc_fp;
    DMPL_free_ft                 DMPL_free_fp;
    DMPL_error_string_ft         DMPL_error_string_fp;
    DMPL_prints_ft               DMPL_prints_fp;
} DMPL_callbacks_t;
```

Fig. 5. DMPL Call Back Table Definition

The DLL must be able to identify the thread or process associated with debugger calls. A call to `DMPL_put_process_info_fp` stores process information in the debugger while `DMPL_get_process_info_fp` retrieves the previously stored process information. The functions `DMPL_put_thread_info_fp` and

DMPL_get_thread_info_fp provide similar functionality for threads. To provide access to process-wide information, the function DMPL_get_process_fp returns the process within which a thread resides.

Several callback functions combine to provide the needed mechanisms to communicate type and data information between the DLL and the debugger. The DLL uses DMPL_get_type_sizes_fp to get fundamental type sizes for a specific process. DMPL_find_symbol_fp provides a mechanism to look up a symbol in a process. Given a type name, the DLL can retrieve the associated information in the process with DMPL_find_type_fp. Data can be read from an address within a specific thread by using DMPL_get_data_fp while the value of a pthread key can be read by using the function DMPL_get_pthread_key_fp. Data can be converted into the host format with DMPL_target_to_host_fp.

```
typedef void (*DMPL_put_process_info_ft)
           (DMPL_process_t *, DMPL_process_info_t *);
typedef DMPL_process_info_t * (*DMPL_get_process_info_ft)
           (DMPL_process_t *);
typedef void (*DMPL_put_thread_info_ft)
           (DMPL_thread_t *, DMPL_thread_info_t *);
typedef DMPL_thread_info_t * (*DMPL_get_thread_info_ft)
           (DMPL_thread_t *);
typedef DMPL_process_t * (*DMPL_get_process_ft)
           (DMPL_thread_t *);
typedef int (*DMPL_get_type_sizes_ft)
           (DMPL_process_t *, DMPL_target_type_sizes_t *);
typedef int (*DMPL_find_symbol_ft)
           (DMPL_process_t *, const char *, DMPL_taddr_t *);
typedef DMPL_type_t * (*DMPL_find_type_ft)
           (DMPL_process_t *, const char *, int);
typedef int (*DMPL_get_data_ft)
           (DMPL_thread_t *, DMPL_taddr_t, void *, int);
typedef int (*DMPL_get_pthread_key_ft)
           (DMPL_thread_t *, DMPL_tword_t, DMPL_taddr_t *);
typedef void (*DMPL_target_to_host_ft)
           (DMPL_thread_t *, const void *, void *, int);
typedef int (*DMPL_sizeof_ft) (DMPL_type_t *);
typedef int (*DMPL_field_offset_ft)
           (DMPL_type_t *, const char *);
typedef void * (*DMPL_malloc_ft) (size_t);
typedef void (*DMPL_free_ft) (void *);
typedef const char * (*DMPL_error_string_ft) (int);
typedef void (*DMPL_prints_ft) (const char *);
```

Fig. 6. DMPL Callback Function Type Definitions

The debugger uses two function types extensively to implement the type and data callback functions. A function of type DMPL_sizeof_ft determines the size of a specific type. For structs and similar types, the DLL can find the field offset pointer of a member name of the type with a DMPL_get_field_offset_ft function.

The remaining callback functions provide important utility operations to the DLL. The DLL can allocate and free debugger memory with DMPL_malloc_fp and DMPL_free_fp. The DLL should not call malloc and free directly. The DLL can

determine the error string associated with an error code by calling `DMPL_error_string_fp`. `DMPL_prints_fp` allows the DLL to print any messages. The DLL should not print messages directly.

4.2.2 DMPL Debugger Functions

The debugger accesses several functions in the DMPL interface. To display the value or address of the thread variable, the debugger will make calls to the DLL to get the absolute address of the object. The DLL also provides initialization, version information and the abilities to release process and thread information and to convert an error code to a string. The rest of this section describes the functions for debugger use, which are defined in Figure 7.

```
extern int DMPL_initialize (const DMPL_callbacks_t *);
extern const DMPL_rtl_names *DMPL_get_rtl_names (int);
extern int DMPL_demangle_name (const char *, int *, char *,
                               int, int *, int *);
extern int DMPL_get_tls_address
           (DMPL_thread_t *, DMPL_taddr_t, DMPL_taddr_t *);
extern int DMPL_destroy_thread_info (DMPL_thread_info_t *);
extern int DMPL_destroy_process_info (DMPL_process_info_t *);
extern const char *DMPL_version_string (void);
extern int DMPL_version_compatibility (void);
extern const char *DMPL_error_string (int);
```

Fig. 7. DMPL Functions Accessed by the Debugger

The debugger calls `DMPL_get_rtl_names` in order to obtain the names of the run time library names invoked by a given OpenMP construct. The parameter of this function is a DMPL language code; as already described, the parameter is passed as an `int` to avoid size conflicts. The return value is a pointer to a struct of two functions:

```
typedef struct {
  const char *main_task_dispatcher;
  const char *microtask_invoker;
} DMPL_rtl_names;
```

The main_task_dispatcher function is the outlined routine called by the main task to invoke the OpenMP parallel region or worksharing construct, which we refer to as a microtask. The second function invokes the actual microtask.

The DMPL interface provides the debugger with scoping functions to demangle mangled names and to locate the actual storage used for thread private variables. The second parameter of the `DMPL_demangle_name` function allows the debugger to allocate more space for the result if it is too long for the initial `demangled_name` buffer. Thread private data can be implemented in various ways: thread local storage system facilities, virtual-address-map page aliasing, and pthread key specific data. Thus, we provide `DMPL_get_tls_address` to obtain the actual address of a thread private object in this thread, given its apparent address in the process or thread.

The remaining functions in the DMPL interface that are used by the debugger provide important utility functions. The debugger releases any process or thread informa-

tion associated by the DLL with a process (or thread) by calling `DMPL_destroy_process_info` (or `DMPL_destroy_thread_info`). The debugger ensures that the DLL provides the expected interface through the versioning functions `DMPL_version_string` and `DMPL_version_compatibility`. Finally, `DMPL_error_string` converts error codes to strings for the debugger.

5 DMPL Example

This section presents an example use of the DMPL interface, based on the short OpenMP program shown in Figure 8 in which the thread private variable `a` counts the number of iterations that the specific thread performs.

```
#include <omp.h>
#include <stdio.h>
int a = 0;

#pragma omp threadprivate(a)
void foo (int i)
{
   printf ("id=%d, i=%d, a= %d\n", omp_get_thread_num(), i, ++a);
   fflush (stdout);
   return;
}

const int tcount = 4;
const int titers = 10;

int main (int argc, char **argv)
{
   int i;
   omp_set_num_threads (tcount);
   omp_set_dynamic (0);
#pragma omp parallel for
   for (i = 0; i < titers; i++)
       {foo (i);}
   return 0;
}
```

Fig. 8. User Code for DMPL Usage Example

```
#include <omp.h>
#include <stdio.h>
int get_thread_obj_address(thread_handle *thr, thrd_addr in_addr,
                           thrd_addr *obj_addr)
{
   int rc;
   …
   rc = DMPL_get_tls_address (thr, in_addr, obj_addr);
   /*  check return code  */
   …
}
```

Fig. 9. DMPL Debugger Routine Example

For debugging purposes, we will assume that the user's intent was for the program to create four threads, each of which executes `foo` ten times, with the print statement labeling the iterations from zero to nine via `a`. The program as written creates four threads, two of which execute `foo` three times, while the other two execute it twice. The iterations of the threads are labeled from one to the thread's execution count. The

```
#include <omp.h>
#include <stdio.h>

/* An implementation of DMPL_get_tls_address for a system in
 * which the address given to the debugger for a thread
 * private variable represents the offset of the variable
 * into storage which is hung on a pthread_get_specific key.
 * This is assuming that it knows the implementation of
 * a pthread key and that it is one word big.
 *
 * A real implementation should cache this value with the thread to
 * avoid all the lookups on each of the calls to this routine.
 */
int DMPL_get_tls_address(DMPL_thread_t *thr,
                         DMPL_taddr_t in_addr,
                         DMPL_taddr_t *out_addr)
{
   int rc;
   DMPL_tword_t buf[2];
   DMPL_taddr_t threads, base;
   DMPL_process_t *process;
   DMPL_tword_t key_val;
   DMPL_target_type_sizes_t ts;

   /* Get the process within which the thread resides */
   process = (DMPL_callbacks->DMPL_get_process_fp)(thr);
   /*  Get the size of the primitive types  */
   rc = (DMPL_callbacks->DMPL_get_type_sizes_fp) (process, &ts);
   /* Find the address where the key value is stored, */
   /* assumed to be on the global variable  "__XYZ_OMP_TLS_key */
   rc = (DMPL_callbacks->DMPL_find_symbol_fp)(process,
                                              "__XYZ_OMP_TLS_key",
                                              &threads);
   /* Read the key value. */
   rc = (DMPL_callbacks->DMPL_get_data_fp) (thr_, threads,
                                            (void *)buffer,
                                            ts.pointer_size);
   key_val = buffer[0];
   /* Read pthread specific value for the key in the thread. */
   rc = (DMPL_callbacks->DMPL_get_pthread_key_fp) (thr_,
                                                   key_val, &base);
   *out_addr_ = base+in_addr;
   return rc;
}
```

Fig. 10. Sketch of Implementation of `DMPL_get_tls_address` Library Routine

error in per thread execution count results from an incorrect understanding of the semantics of `omp parallel for`, while the labeling error results from using the prefix operator instead of the postfix operator.

Although the errors are easily understood in this example, a debugging session could often involve the user examining the values of `a` in each thread. Figure 9 shows a portion of the steps that the debugger executes to support this examination. We assume the debugger has already called `DMPL_initialize` to load and to initialize the DLL library and that it has used `DMPL_demangle_name` to determine that the object of interest is the thread private variable `a`. The debugger then determines the true location of thread `thr`'s version of `a` by calling the library routine `DMPL_get_tls_address` with `in_addr`, the apparent address of `a`.

A sketch of an implementation of the DLL `DMPL_get_tls_address` routine is provided in Figure 10. As discussed previously, compilers implement thread private variables through widely varied methods. In our sketch, we assume that the address the compiler gives to the debugger for a thread private variable represents the offset of the variable into storage accessed through a pthread_get_specific key. Our sketch uses several of the callback routines registered by the debugger through `DMPL_initialize`. First, the process-specific information is retrieved via `DMPL_get_process_fp`. Next, the type sizes of the process are determined by calling `DMPL_get_type_sizes_fp` and the address of the key value is looked up with `DMPL_find_symbol_fp`. Having located the key, the routine uses `DMPL_get_data_fp` and `DMPL_get_pthread_key_fp` to retrieve its value. Finally, the actual address is calculated and returned to the debugger.

6 Discussion

Although standards exist for OpenMP directives, there is currently no standard for the information that compilers or other tools require to present information consistently with the OpenMP programming model. Since compilers implement OpenMP directives differently, compilers should adopt DMPL as a standard interface for providing debuggers and similar tools that information.

The POMP interface has already been proposed for providing information to performance tools. A legitimate question is whether performance tools and debuggers could be properly served by a unified information interface. However, the interfaces work in fundamentally different ways - the performance interface works within the application while the debugger interface is external. Compiler, run-time library and tool implementers have agreed during meetings of the OpenMP Tools Committee that they prefer two interfaces. The compiler writers and run-time system implementers have committed to providing two interfaces if the committee adopts them.

Even though the access mechanisms are sufficiently different to justify two standard information interfaces, there is significant intersection of the information needed for them. For example, DMPL includes functions to demangle names. Name deman-

gling is also useful to performance tools. For this reason, we anticipate a set of low-level information standards or vendor supplied tools, such as a name demangler.

DMPL is a basic interface for the OpenMP debugging DLL. Implementations of this library with slightly different naming conventions are already available from IBM and Intel for its Guide compilers. OpenMP debugging on those platforms with current versions of TotalView demonstrates that DMPL provides significant support for the OpenMP programming model [4]. However, DMPL is an evolving interface and we recognize that additional functions and types may be needed. If this occurs, the OpenMP Tools Committee will create a new version of this document to fill that need.

References

1. OpenMP Architecture Review Board: OpenMP Fortran Application Program Interface, Version 2.0. OpenMP Architecture Review Board (2000)
2. OpenMP Architecture Review Board: OpenMP C and C++ Application Program Interface, Version 2.0. OpenMP Architecture Review Board (2002)
3. Mohr, B., Malony, A.D., Hoppe, H.C., Schlimbach, F., Haab, G., Hoefflinger, J., Shah, S.: A Performance Monitoring Interface for OpenMP. In Proceedings of the Fourth European Workshop on OpenMP (EWOMP 2002). Rome (2002)
4. Etnus LLC: TotalView Reference Guide, Version 6.0. Etnus LLC (2002)
5. Cownie, J., Gropp, W.: A Standard Interface for Debugger Access to Message Queue Information in MPI. Sixth European PVM/MPI Users' Group Meeting. (1999)
6. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, Institute for Defense Analysis, Center for Computer Sciences, Bowie, Maryland (1999)

Appendix: Dmpl.h Header File

This section provides the full contents of the dmpl.h header file.

```
#ifdef __cplusplus

extern "C" {
// }
/* That curly just stops the emacs indenter getting confused and
   indenting everything */

#endif

/*****************************************************************
 * This file contains the definition of the interface between a debugger
 * and a DLL which can be loaded into the debugger which contains
 * knowledge of the parallel run time used by some compiler.
 *
 * The interface is designed to allow the debugger to present to thread
 * local variables (such as OpenMP OMP THREADPRIVATE common blocks or
 * variables).
 *
 * Such global or file scoped variables must be tagged in the normal
 * debug information (DWARF, STABS, COFF,...) so that the debugger knows
 * that they are thread private and require special treatment.
 *
 * For a THREADPRIVATE common block, we would expect the normal
 * definition of the common block, but the base address to be tagged
 * as thread private. Such a common block should be declared in
 * function scope as is usual for a Fortran common block.
 *
 * When called upon to display the value (or address) of a
 * THREADPRIVATE object the debugger will make calls down into the
 * dynamic library to extract the absolute address of the object.
 *
 * The dll will in turn make calls back into the debugger if it
 * requires access to information about (or contained in) the target
 * process (and thread).
 *
 * The debugger may be debugging multiple independent processes
 * simultaneously, which can be executing different programs and have
 * different execution models (for instance 32 bit and 64 bit
 * processes). Therefore the dll must not make use of global data
 * internally, but rather associate any data it requires to maintain
 * between calls with the specific object in question.
 *
 * Callbacks are provided to the debugger to perform the following
```

```
* operations :-
*
*    Miscellaneous operations
*      allocate store
*      release previously allocated store
*      print an error message
*
*    Operations related to a process
*      associate data with the process
*      get back data previously associated with the process
*      determine the size of fundamental target types
*      lookup named types
*      get the size of a type
*      get field offsets within a type
*      lookup the address of a global symbol
*
*    Operations related to a thread
*      associate data with the thread
*      get back data previously associated with the thread
*      get the process within which the thread lives
*      read store given an absolute address
*      convert data from target format to host format
*         (e.g. byte flipping)
*      look up the value of a pthread key
*
* The dll provides the debugger with functions to perform
* the following operations :-
*
*    Miscellaneous
*      basic initialization
*      get version compatibility
*      get version string
*      convert an error code into a string
*
*    Operations associated with a process
*      release any store associated by the dll with the process.
*
*    Operations associated with a thread
*      release any store associated by the dll with the thread.
*      return the address of a thread private object given
*       the "address" describing the object from the debug information.
*
* While it would be possible to separate some of the operations on
* processes into operations on an executable image, (which would
* perhaps allow more optimization in the case where a single image
* is being used in many processes), we choose not to do that, since
* global data objects from a single image may be mapped at different
* addresses in different processes if they are actually asociated
* with a dynamic library, and that may well be the case for some
* of the objects of interest.
```

```
 */

#define XLD_INTERFACE_COMPATIBILITY 1 /* Version of this interface */

/*****************************************************************
 * Types which form part of the interface
 *
 * Since we can already see 64bit processes coming at us we define the
 * interface to handle them, and pay a small cost when debugging 32 bit
 * processes.
 */

/* Type big enough for an address */
typedef unsigned long long DMPL_taddr_t;

typedef long long          DMPL_tword_t;

/* A structure for (target) architectural information */
typedef struct
{
  int short_size;                       /* sizeof (short) */
  int int_size;                         /* sizeof (int) */
  int long_size;                        /* sizeof (long) */
  int long_long_size;                   /* sizeof (long long) */
  int pointer_size;                     /* sizeof (void *) */
} DMPL_target_type_sizes_t;

/*****************************************************************
 * Opaque types which are used to provide type safety across the
 * interface.  We could use "void *" for these, but having these
 * undefined structures provides more compile time checking (at the
 * cost of having to have explicit casts in the library and support
 * code).
 */

/* Types which will be (cast to) concrete types in the debugger */
typedef struct _DMPL_process_t DMPL_process_t;
typedef struct _DMPL_thread_t  DMPL_thread_t;
typedef struct _DMPL_type_t    DMPL_type_t;

/* Types which will be (cast to)  concrete types in the DLL */
typedef struct _DMPL_process_info_t DMPL_process_info_t;
typedef struct _DMPL_thread_info_t  DMPL_thread_info_t;

/* Result codes.
 * DMPL_ok is returned for success.
 * Anything else implies a failure of some sort.
 *
 * Most of the functions actually return one of these, however to avoid
 * any potential issues with different compilers implementing enums as
```

```
 * different sized objects, we actually use int as the result type.
 *
 * Note that both the DLL and the debugger will use values starting at
 * DMPL_first_user_code, since you always know which side you were
 * calling,this shouldn't be a problem.
 *
 * See below for functions to convert codes to strings.
 */
enum
{
  DMPL_ok = 0,
  DMPL_tls_unallocated, /* No space has been
                           allocated for this object */
  DMPL_name_too_long,   /* Return from demangle, buffer is too small */
  DMPL_name_unchanged,  /* Return from demangle name wasn't demangled */
  DMPL_first_user_code = 100  /* Allow for more pre-defines */
};

/*************************************************************************
 * Values for languages.
 */
enum
{
  DMPL_lang_unknown = 0,
  DMPL_lang_c       = 'c',
  DMPL_lang_cplus   = 'C',
  DMPL_lang_f77     = 'f',
  DMPL_lang_f9x     = 'F'
};

/* Demangling info codes.
 * DMPL_varinfo_needs_dereference is a bit which can be or-ed into any of
 * the other codes.
 */
enum
{
  DMPL_varinfo_needs_dereference = 0x8000, /* needs extra dereference */
  DMPL_varinfo_none              = 0,      /* no info */
  DMPL_varinfo_private,                    /* was on private list */
  DMPL_varinfo_shared,                     /* was on shared list */
  DMPL_varinfo_firstprivate,               /* was on firstprivate list */
  DMPL_varinfo_lastprivate,                /* was on lastprivate list */
  DMPL_varinfo_firstlastprivate,           /* was on first and */
                                           /* last private lists */
  DMPL_varinfo_reduction,                  /* was on reduction list */
  DMPL_varinfo_threadprivate,              /* was on threadprivate list */
  DMPL_varinfo_threadshared,               /* was on threadshared list */
  DMPL_varinfo_copyin                      /* was on copyin list */
};
```

```
/*****************************************************************
 * (Types for) Callback functions provided by the debugger to the DLL.
 */

/*****************************************************************
 * Miscellaneous functions.
 */
typedef void *       (*DMPL_malloc_ft) (size_t);
typedef void         (*DMPL_free_ft)   (void *);
typedef void         (*DMPL_prints_ft) (const char *);
typedef const char * (*DMPL_error_string_ft) (int);

/*****************************************************************
 * Process specific functions.
 */

/* Store information on the process */
typedef void (*DMPL_put_process_info_ft)
             (DMPL_process_t *, DMPL_process_info_t *);

/* Get it back */
typedef DMPL_process_info_t * (*DMPL_get_process_info_ft)
                              (DMPL_process_t *);

/* Get the fundamental type sizes for this process */
typedef int (*DMPL_get_type_sizes_ft)
            (DMPL_process_t *, DMPL_target_type_sizes_t *);

/* Given a process look up the specified symbol */
typedef int (*DMPL_find_symbol_ft) (DMPL_process_t *,
                                    const char *, DMPL_taddr_t *);

/* Given a process look up the specified type */
typedef DMPL_type_t * (*DMPL_find_type_ft) (DMPL_process_t *,
                                            const char *, int);


/*****************************************************************
 * Type specific functions.
 */

/* Given a type find its size */
typedef int (*DMPL_sizeof_ft) (DMPL_type_t *);

/* Given a type and a member name find the field offset */
typedef int (*DMPL_field_offset_ft) (DMPL_type_t *, const char *);
```

```
/*************************************************************************
 * Thread specific functions.
 */

/* Store information on the process */
typedef void (*DMPL_put_thread_info_ft) (DMPL_thread_t *,
                                         DMPL_thread_info_t *);

/* Get it back */
typedef DMPL_thread_info_t * (*DMPL_get_thread_info_ft) (DMPL_thread_t *);

/* Get the process within which the thread resides */
typedef DMPL_process_t * (*DMPL_get_process_ft) (DMPL_thread_t *);

/* Read data from a thread */
typedef int (*DMPL_get_data_ft) (DMPL_thread_t *,
                                 DMPL_taddr_t, void *, int);

/* Convert data into host format */
typedef void (*DMPL_target_to_host_ft) (DMPL_thread_t *,
                                        const void *, void *, int);

/* Get the value of a pthread key */
typedef int (*DMPL_get_pthread_key_ft) (DMPL_thread_t *,
                                        DMPL_tword_t, DMPL_taddr_t *);


/*************************************************************************
 * The callback table itself
 */

typedef struct
{
  DMPL_put_process_info_ft     DMPL_put_process_info_fp;
  DMPL_get_process_info_ft     DMPL_get_process_info_fp;
  DMPL_put_thread_info_ft      DMPL_put_thread_info_fp;
  DMPL_get_thread_info_ft      DMPL_get_thread_info_fp;
  DMPL_get_process_ft          DMPL_get_process_fp;
  DMPL_get_type_sizes_ft       DMPL_get_type_sizes_fp;
  DMPL_find_symbol_ft          DMPL_find_symbol_fp;
  DMPL_find_type_ft            DMPL_find_type_fp;
  DMPL_get_data_ft             DMPL_get_data_fp;
  DMPL_get_pthread_key_ft      DMPL_get_pthread_key_fp;
  DMPL_target_to_host_ft       DMPL_target_to_host_fp;
  DMPL_malloc_ft               DMPL_malloc_fp;
```

```
  DMPL_free_ft                  DMPL_free_fp;
  DMPL_error_string_ft          DMPL_error_string_fp;
  DMPL_prints_ft                DMPL_prints_fp;
} DMPL_callbacks_t;


/*****************************************************************
 * Calls provided by the DLL to the debugger.
 */

typedef struct
{
  const char * main_task_dispatcher; /* Name of the function which the
                                      * main task calls to invoke
                                      * a microtask. The routine which
                                      * called this is the main task.
                                      */
  const char * microtask_invoker;    /* Name of the function which invokes
                                      * the microtask. The function called
                                      * by this is the microtask
                                   */
} DMPL_rtl_names;

extern int DMPL_initialise (const DMPL_callbacks_t *);
extern int DMPL_version_compatibility (void);
extern const char * DMPL_version_string (void);
extern const char * DMPL_error_string (int);

extern int DMPL_destroy_process_info (DMPL_process_info_t *);
extern int DMPL_destroy_thread_info  (DMPL_thread_info_t *);

/* Demangle a mangled name, returns the demangled name
 * and its length, if the demangled name would be too long
 * for the buffer, the desired length should still be returned so that
 * the debugger knows how much space to allocate for the result.
 */

/*                              mangled_name, res_len, demangled, buf_len, */
extern int DMPL_demangle_name (const char *, int *,   char *,    int,
                               int *, int *);
/*                              flags, decl_line */

/* Given an address in the thread (process), which the debugger has
 * been told via the debug information (stabs) represents a thread
 * private object, obtain the actual address which refers to that
 * object in this thread.
 */
extern   int   DMPL_get_tls_address   (DMPL_thread_t   *,   DMPL_taddr_t,
  DMPL_taddr_t *);
```

```
/* Return the names of the appropriate routines in the run time, given
 * the target language */
extern const DMPL_rtl_names * DMPL_get_rtl_names (int);

#ifdef __cplusplus
} /* End of extern "C" */
#endif
```